Contents

Introduction **2**

What is Functional Testing? **2**

Automated functional testing **3**

Critical Success Factors *3*

Abbot Java GUI Test Framework **3**

Building functional tests (SwingUnit) **4**

The Abbot Framework *4*

Robot (the user event generator, and the root of all Testers) 4

ComponentLocation 4

ComponentReferences 5

Scripts 5

Other Features 5

Goals of the Abbot Framework *5*

Abbot helps you test your Java UI. **6**

Conclusion **7**

Appendix A

Appendix B

Introduction

Functional testing assures that our implementation of SAP meets your business requirements. Given the highly configurable and tightly integrated nature of the SAP modules, as well as the probabil- ity that you will also integrate in-house applications or third-party plug-ins, it is a critical and challenging task requiring the verification of hundreds or even thousands of business processes and the rules that govern them.

We have tried to describe the functional testing and abbot-1.0.2 as testing the interfaces written in java testing frame work.

What is Functional Testing?

Test automation is not a panacea, but it can make a dramatic difference in the quality and stability of your SAP deployment over the long term. The key is to understand when automation works and when it does not, and how to assure your success.

"The objective of function test is to measure the quality of the functional (business) components of the system". Tests verify that the system behaves correctly from the user / business perspective and functions according to the requirements, models, storyboards, or any other design paradigm used to specify the application. The function test must determine if each component or business event: performs in accordance to the specifications, responds correctly to all conditions that may be presented by incoming events / data, moves data correctly from one business event to the next (including data stores), and that business events are initiated in the order required to meet the business objectives of the system.

Application functional testing is the most widely accepted testing practice in software development organizations today, and hundreds of millions of dollars have been spent on automated tools to support the practice. Yet, organizations still struggle to achieve software quality and suffer inconsistent, inefficient, and often inaccurate application of the practice. Over 70 percent of software testing today is still done manually despite the investment in automation. Understanding what has or hasn't been tested is still most organizations' major challenge, and the time commitment required for QA functional testing is increasingly one of organizations' greatest concerns for meeting application delivery and time-to-market demands.

Without a well-defined and implemented "best" practice, functional testing will continue to be practiced incorrectly and suffer inaccurate or, at best, incomplete results. Automated tools alone cannot ensure that functional testing is complete and accurate. Achieving this goal requires that a tool be supported by a well-defined practice for its use and be integrated into an organization's software development process. The necessary level of commitment and practice maintenance is often underestimated and is typically the reason for poor results and the abandonment of automated tools in favor of manual testing. An effective and consistently-applied functional testing practice is the only way to ensure that software satisfies expected functionality and reliability objectives.

Software development organizations with an effective functional testing practice have a fast and objective way to determine whether each functional requirement is actually implemented in the code. With functional testing, the team translates functional requirements into executable test cases that confirm how well the code satisfies the requirements at any given time. It provides unprecedented objective insight into requirement status and prevents the missing or incorrect functionality implementations that can lead to countless rewrites (and then budget overruns and missed deadlines), user dissatisfaction, and project failure.

An effective functional testing practice should be a natural extension of a requirements management policy. Ideally, a requirements management policy describes how to build effective use cases for the features expected in the current release, and how to map those use cases to verifiable test cases. A test plan is then developed and implemented to build a suite of executable tests that frame and verify the functionality requirements, providing a

fast and objective way to assess the status of expected functionality. These tests can then be executed regularly to ensure that code modifications do not unintentionally change previously verified functionality.

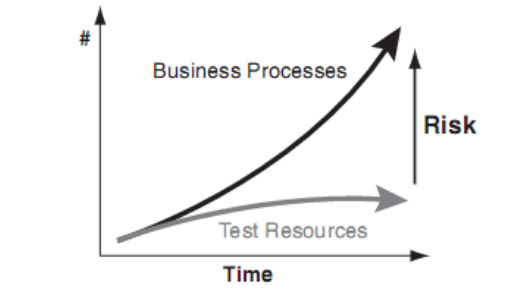
Automated functional testing

Test automation is not a panacea, but it can make a dramatic difference in the quality and stability of your SAP deployment over the long term. The key is to understand when automation works and when it does not, and how to assure your success.

There are three key benefits to automation:

1. Expand your test coverage.
2. Save time and resources.
3. Retain knowledge.

This repeatability saves time and resources as well. Instead of requiring repetitive manual effort to revivify processes each time changes are introduced, tests can be automatically executed in an un- attended mode. This allows your resources to focus on adding new tests to support new functionality instead of constantly repeating existing tests.



Test Workload Compared to Test Resources

Ad hoc tests are uniquely suited to manual testing because they require creativity and are deliberately unpredictable. By allowing automation to take care of what you expect to work, you can free your experts to try to break the system.

Critical Success Factors

- Successful test automation requires:
- Management commitment
- Planning and training
- Dedicated resources
- Controlled test environment
- Pilot project

Abbot Java GUI Test Framework

Abbot is a simple framework for unit and functional testing of Java GUIs. Facilitates generating user actions and examining component state. Supports recording and playback on any Java application.

Building functional tests (SwingUnit)

The Abbot Framework

Abbot provides a framework for testing your GUI regardless of the current state of your code. If you are doing test-first development with lots of unit testing, then Abbot can provide the developer the tools needed to write individual unit tests. If you have an existing code base without existing unit tests, you can use the scripting level of Abbot to start building functional test scaffolding around your application until it is sufficiently stable to support refactoring and addition of unit tests.

In general, testing with Abbot consists of getting references to GUI components and either performing user actions on those components or making some assertions about their state. To facilitate this process, the framework provides ComponentReferences to get a handle on a GUI component (even when it may not yet exist), and extended Robot-like objects, which know how to perform user-level actions on various GUI components. These operations may be done from either a high-level script (useful for functional/acceptance testing) or directly from Java code (for example in a JUnit TestCase method).

Robot (the user event generator, and the root of all Testers)

This is the fundamental component of Abbot, "A Better 'Bot". It provides basic component-level events upon which more complex tester objects can be built. In the absence of a specific tester object for a given component, the events provided by the Robot class are sufficient to reconstruct any user actions.

For each Component to be tested, there is a Tester class which provides user actions and tests (assertions) specific to that component. This is the class to extend when you need to provide access to a custom GUI component. The base class, abbot.tester.ComponentTester, provides default actions for most common user actions, including clicking a mouse button, selecting from a menu, and typing text. The tester system is easily extensible to provide more specific actions or property lookup for custom components; for example, JTableTester provides an action which selects the cell at a given row, column location. JPopupMenuTester provides a method which returns a list of all the menu items currently on a popup menu.

ComponentLocation

Abbot represents locations within a given component with a dedicated data type, the ComponentLocation. This means that any component with internal substructure (cells in a table, rows in a tree, items in a list, etc) may identify a substructure location with a class derived from ComponentLocation. For example, a JTableLocation may be constructed with a Point (raw coordinates), a [row, col] pair, or a String representation of the value in the desired cell. Using these location types avoids a proliferation of API methods, so you can simply have one method per type of user action, like this:

```
public void actionClick(Component, ComponentLocation);
public void actionDrag(Component, ComponentLocation);
public void actionShowPopupMenu(Component, ComponentLocation);
```

instead of many like this:

```
public void actionClick(Component c, Point where);
public void actionClick[Cell](JTable c, int row, int col);
public void actionClick[Row](JTree c, int row);
public void actionClick[Value](JTable c, String value);
// Repeat for actionDrag, actionDrop, actionShowPopupMenuAt, etc
```

From a script, these are automatically generated as needed; in general you can use (x,y), [index], [row,col], or "value" notations depending on how you want to refer to a particular substructure element. When recording, the `String` value of a location will be used by default.

ComponentReferences

ComponentReferences allow you to use a few known attributes of a component to refer to that component, whether or not the component currently exists or is visible. The framework provides automatic lookup, so that you don't have to clutter your object API with accessors like getOKButton and getCancelButton. When tests are run, component references may be converted to the actual Component they represent just in time, at the point where the component itself is actually needed. Component lookup is fuzzy, such that minor changes in attributes (such as position, parent or size of a component) generally won't break the script.

Scripts

Scripts are the basic units of test execution in the Abbot framework. In general these consist of component lookup information, component actions (such as menu selections, typed text, etc.), and assertions regarding the GUI's state. Scripts for unit tests may be very simple, while scripts for functional testing may be far more lengthy and complex. See the abbot.script package for more details.

When using a script, all information required to run the test is encapsulated within the script itself, to provide independence from any particular testing environment. This facilitates running the scripts under either the script editor, JUnit, or some other harness.

The ScriptFixture provided for use with JUnit derives from TestCase and is a very simple wrapper to adapt the script to the JUnit testing framework. Scripts may be grouped into a test suite by using the ScriptTestSuite class, which can automatically collect a group of tests into a suite based on file hierarchy, filenames, or other criteria. The preferred method of creating a suite is to derive your test class from ScriptFixture, then have the static "suite" method for that derived class create an instance of ScriptTestSuite which selects for the scripts you wish to group. See the ScriptTestSuite API documentation for details.

Other Features

GUI Test Fixture

A `ComponentTestFixture` is provided for testing GUI components and groups of components directly from Java code. This fixture extends `TestCase`, so you can manually write unit tests using any of the facilities of the framework.

This fixture takes care of providing a clean environment for instantiating, operating on, and examining GUI components, which includes properly disposing of all instantiated components at the end of each test. This avoids the tediousness of having to keep track of everything you displayed and tracking down anything that might have been displayed that you didn't know about (common when testing already-developed code). It also provides methods for looking up desired components in the hierarchy.

Editor (Costello)

A script editor is provided to facilitate creation and maintenance of Abbot scripts. The editor supports live recording of user events to facilitate writing scripts, and can also play them back for debugging.

Recorders

Recorders provide a means for the script editor to capture high-level semantic information instead of basic mouse and key events. Recorders are provided for most standard Swing components, but events will be captured properly even in the absence of a customized Recorder for a given component. The editor supports plug-in recorders to support additional custom components.

Goals of the Abbot Framework

Aside from the primary goal of reliably automating tests of GUIs and GUI components, the Abbot framework has the following goals:

- *Reliable reproduction of user input for tests*
  One reason GUIs don't get tested is because it's usually not easy for a developer to simulate user input. Problem solved, no more excuses for not unit testing your GUI components.
- *Scripted control of actions and inspection*
  Rather than writing new code which requires compilation, the test scripts are dynamically interpreted, which is more appropriate for this level of tests. All functionality, of course, is available for direct invocation in Java unit test code.
- *Loose component bindings*
  GUI layouts tend to change over time. A raw event recorder/playback mechanism will likely break if you're operating in absolute coordinates; different platforms generally size their components differently, so tests may not work cross-platform. Abbot for the most part doesn't care where components are positioned, as long as it can look them up dynamically with a few tagging identifiers.
- *Specify high-level semantic actions, but use low-level OS events to implement them.*
  The java.awt.Robot class is capable of directly reproducing user events. Unfortunately, it is far too basic to be easy to use for unit tests. Abbot builds a level of abstraction on top of java.awt.Robot, mirroring the way Swing Components are a level of abstraction above basic windows. At this level, tests are much easier to maintain since you can deduce from a glance what a set of instructions will do.
- *Support live recording of high-level semantic events.*
  While you can always write XML by hand, or edit each script step individually, it makes more sense to leave it to the computer to do the grunt work. The Costello editor can automatically record a sequence of user actions and save them to a script.

Abbot helps you test your Java UI.

Abbot helps you test your Java UI. Abbot framework comprises Abbot, which lets you programmatically drive UI components, and Costello (built on Abbot) which allows you to easily launch, explore and control an application. The framework may be used with both scripts and compiled code.

This overview of the Abbot system will give you an idea of the different system components and how it is used. There is also some background concerning the issues involved with Java GUI testing.

You can see what Costello (the script editor) looks like or try it out via Java Web Start by using the links to the right. The screen shot shows a sample application, its UI hierarchy, and some component properties as displayed by Costello. Here is what GUI unit test code looks like using Abbot (see Tutorial 1 or Tutorial 2 for more examples).

Refer to the API documentation for the ComponentTestFixture, ComponentTester and Robot classes, as well as the framework tests in the test source subdirectory. You can also record user actions directly into a script which controls the event playback and testing.

This form of test is more suitable to integration/functional testing. The Costello script editor is provided which can record user actions and facilitate script construction and maintenance. It also provides insight into the hierarchy of the application under test.

You can launch the editor with the Web Start link above and immediately point it at any application you wish to launch and examine. The editor provides a hierarchy browser which displays all components in use by the application, as well as a wealth of information about any component selected in the hierarchy.

JUnit is used as the controlling harness for running tests and suites of tests, but scripts can easily be wrapped to run in other environments. Scripts may also be used to create a demo or tutorial for an application. You can drive the actual application and provide popup annotations while it is running.

Conclusion

An effective functional testing practice involves the definition of guidelines for using functional testing technologies effectively, and then the implementation and integration of those guidelines (along with supporting technologies and configurations) into your software development lifecycle to ensure that your teams apply the policy consistently and regularly. It also requires a means to monitor and measure the policy's application, as well as report the data it is tracking.

To achieve effective black box unit testing, you must not only have a defined practice for its use, but that practice must be implemented and integrated into your software development lifecycle so that it is used consistently and regularly across your software development team. It is also important to have the means to monitor and measure its use.

Appendix A: User Manual of Abbot-1.0.2

This document provides more in-depth descriptions of how to use the Costello script editor. You should also run the script "editor-tutorial-1.xml", which follows the information included here but displays it in the context of the editor application itself.

```
% java -cp lib/costello.jar junit.extensions.abbot.ScriptFixture doc/editor-tutorial-1.xml
```

Creating a Script

Scripts are a convenient way to organize a suite of actions and assertions that make up a test case. They are saved to disk as XML as a convenient and standardized storage format.

Scripts are primarily composed of actions, assertions, and component references. ***Actions*** are things that a user usually does to operate a GUI component, like pushing a button, selecting a menu item, or entering text. Most actions require one or more arguments, one of which is the target GUI component. The Abbot framework allows you to use simple IDs in place of looking up a Java reference to a Component object instance.

Component references represent actual GUI component instances within your code under test. Each has a unique ID that may be used to indicate to other script steps the actual component on which you wish to operate. For example, there might be a component reference for the OK button in a dialog, with an ID of "OK Button". You can then use the string "OK Button" as an argument to an Action in place of the actual button instance.

Finally, ***assertions*** allow you to verify the state of your GUI. You can check the value of GUI component properties or evaluate arbitrary expressions. Normally the script will stop running and report an error if these assertions fail. Assertions may also be changed to wait steps, where the script execution pauses until the condition becomes true.

There are a few other lesser-used Script Step types; see the abbot.script package for more details. In general, scripts should be kept as simple as possible; advanced conditionals and looping constructs do not lend to this intent.

Create New Script

Select **New** from the **File** menu, and you will be prompted for a destination filename. New scripts automatically have Launch and Terminate steps included.

Open an Existing Script

Select **Open** from the **File** menu, and you can browse for an existing script to open. Scripts do not require a filename extension, although ".xml" is used by convention.

Launching an Application

Launch Information

The ***Launch*** Step allows you to indicate the main class name of your code under test. It doesn't have to be "main", but the method must be a public static method. You also must indicate a classpath to be used to find your main class. If you use relative path names, then your scripts will be runnable on different platforms.

If the method you call to launch your code under test does not return, then you must set the Launch step to be threaded. In the editor, there is a checkbox for this option. If the Launch is not threaded, the editor will wait for the method invocation to finish before continuing. You should also use the threaded option if your application displays a modal dialog (or any other GUI which requires interaction) before returning from its main method.

Class Loading

It is preferable to enter the classpath in the Launch step (rather than specifying it in the environment CLASSPATH or on the command-line when launching the editor) so that your classes can be reloaded on each test run. However, some applications improperly use resource or class loading methods that directly access the

system class loader (you should almost never explicitly reference the system class loader, but instead either use the Thread context class loader or the class loader for the current class). In these instances it may be necessary to rely on either the environment CLASSPATH or java.class.path as passed to the editor. Keep in mind that in these situations, running a test the first time may have different behavior from subsequent runs.

Wait for...

The first thing you should do after your application launches, is to wait for it to show up. There's no sense in starting to click buttons that aren't there yet. The most common thing to wait for is your main component window; if you're recording for the first time, this step will automatically be inserted for you (Wait for Component Showing). Some applications may not be ready for input until a splashscreen goes away, or until some icon appears in the toolbar. You should adjust the wait step accordingly.

Duplicate Fixture

You often need to use the same application launch setup for several scripts. The menu option **File->Duplicate Fixture** copies the Launch/Wait for.../Terminate steps into a new script, ready to be filled in.

What goes into a Script?

The **Insert** menu allows you to manually insert any available script step type.

Recording/Playback

The easiest way to see the types of things that go into a script is to start recording. Once you've set up your launch step properly, you can start recording from the **Capture** menu, or with a keyboard shortcut. If your app has not been launched, it will be automatically, and the editor will begin recording your actions. When you are finished recording, you can either activate the editor window or use the same keyboard shortcut to stop recording. Now you can use **Test->Run** or click the **Run** button to play back the recording. Note that recording doesn't know anything about what you're trying to test; you may want to remove extra waits, actions or events that are not pertinent to what you're focusing on.

Assertions

Once you have the basic actions filled in, you'll want to insert some verification about the state of the GUI. You can select components in your application by moving the pointer over them and pressing shift-F1. This will select the given component in the hierarchy browser in the lower left of the editor window. All the properties available for that component will appear in the table to the right of the hierarchy browser. You can select any property and insert an assert step into the script that indicates the script should fail if the given property does not match its expected value. Holding the shift key inverts the sense of the test, and holding down the shortcut modifier (Control on w32, Command on OSX) makes the assert a wait step instead. In most cases, you can replace the expected value of an assertion with a regular expression bounded by slashes, e.g. "/My Application.*/" would match any string that started with "My Application".

You can't interact with the Costello interface while your application under test is showing a modal dialog. If you wish to make an assertion on a component within a modal dialog, you will need to manually make a reference to that component (alt-shift-F1) while the modal dialog is showing, and then dismiss the dialog. You can then select the new reference in the hierarchy browser, and make assertions as usual. Note that the property values you see in this case may not accurately reflect the values in the component when it is showing.

Samples

You can sample any property value for later use by using a Sample step in place of an assert step. You insert a Sample as you would an Assert. You then need to edit the step to identify the name you want to give the property. You can then refer to that value later in your script by using the dereference notation "`${property.name}`", where "`property.name`" is the name you gave the property.

Action Arguments and Methods

The available selection of actions may change based on the currently selected component (shift F1). The **Insert->Action** menu is dynamically generated based on the action methods provided by the appropriate ComponentTester class for the selected component.

When you insert an action, you will be prompted for the arguments to the method invocation. The first argument will often be the ID of the target component (which may be filled in automatically). You should enter the arguments you'd like to use, separated by commas. Once the step is inserted, you can further edit its attributes in the step editor.

Component References

Whenever you reference a component in the GUI hierarchy, a component reference is created for it. This allows you to use its string ID to refer in place of the actual GUI object. Component references keep a set of attribute values which help to look up the actual GUI object when the test is run. These attribute values can be edited to use regular expression values in case the attribute is variable.

Arbitrary Expressions

The Expression step lets you evaluate arbitrary Java expressions should you need to go beyond simple actions and assertions. The expression interpreter is BeanShell, and it has a lot of nifty convenience features too numerous to list here. The interpreter state is maintained throughout a given script run, so one expression can set variables for use by a later expression. See the BeanShell Documentation for details.

Several built-in variables are provided for convenience:

- `finder` a ComponentFinder for looking up components
- `resolver` the current Resolver context
- `script` the containing script

Several built-in methods are provided as well:

- `find(matcher)`
- `find(root,matcher)` returns the first component matching the given spec
- `component(id)` returns the component matching the component reference with the given id
- `reference(id)` returns the component reference with the given id

The standard JUnit `assertX()` methods are provided, and may be invoked directly from the expression, e.g.

- `assertTrue("Component not wide enough", component("myframe").getWidth() > 500);`
- `assertEquals("Wrong number of Frames", 5, Frame.getFrames().length);`

Cleaning Up

Terminate

The terminate step causes all GUI components generated by your code under test to be disposed and made ready for garbage collection. Whether they actually get GC'd will depend on your VM and how well your application cleans up after itself.

Unnecessary Script Steps

Not all recorded steps will be necessary for the test you are writing. Every step should have a purpose, and those that don't should be removed.

Advanced Topics

Compiled Suites of Scripts

You can aggregate a number of related scripts into a suite launchable by the JUnit harness in a number of ways. The class junit.extensions.abbot.ScriptFixture's main routine takes any number of filename arguments and automatically runs them together as a suite of tests. You can extend the ScriptTestSuite class to build a suite of tests based on a customized file system search for scripts. If such a class is found in the editor's classpath, it will be available from the **Select Test Suite...** button and you will be able to flip among all the scripts contained within that suite. An example of this is provided with the sample **MyCode** scripts.

Regular Expressions

In most cases, you can replace any string that represents an expected value with a regular expression bounded by slashes, e.g. "/My Application.*/" would match any string that started with "My Application". This works in component reference attributes, assertion expected values, and other places.

Component Locations

Some component actions take a ComponentLocation argument. These are special objects which encapsulate a component-relative location, such as a cell in a table, a row in a tree, or an item in a list. They usually can be created from a Point (e.g. "(x, y)"), an indexed location (e.g. "[row, col]"), or a quoted value (e.g. "list item" - quotes are part of the argument). Usage of component locations avoids an explosion of APIs that only differ based on whether the target location is a point, and indexed location, or the location of a contained value.

Conversion of Call Arguments

The Abbot parser attempts to convert string arguments into their target argument types, but is not restricted to basic data types to do this. In addition to the parsers in the abbot.script.parsers package, the following conversions can be made:

- Arrays, e.g. "[an,array,of,strings]" is a four-element array
- Properties, e.g. "`${property.name}`", which will be replaced by a sampled property or a system property if no such sampled property exists.
- Component Locations, e.g. "[row]", "(x,y)", "value in list"

Editing Component References

You can edit reference attributes in the component browser by selecting a given reference and changing the attribute values which appear to the right. Be careful when changing the component reference ID, since it is likely used in your script. Those references to the ID will not be automatically changed to the new value.

Appendix B: A simple Presentation of Unit test and JSunit